DATA FLOW ANALYSIS (INTRAPROCEDURAL)

Neil D. Jones
DIKU, University of Copenhagen (prof. emeritus)
Book: NNH = Nielson, Nielson and Hankin Principles of Program Analysis

Slides: downloadable from course home page.

Reading for lectures 2 and 7 December:

► Read and understand NNH sections 1.1, 1.2, 1.3, 1.7, 1.8.
► Skim 1.4, 1.5, 1.6.
► Read and understand NNH section 2.1.

The compiler construction course project may have some application-oriented work based on Chapters 1 and 2.
How is program analysis done?

▶ Many people: decades of practical experience in writing compilers
  (though correctness issues are rarely addressed by compiler hackers)

▶ Engineering methodology: program analysis by fix-point computations.

▶ This was developed by informal, pragmatic, ad hoc methods from the
  1950s called data flow analysis.

Semantics-based program analysis:

▶ Methods formally based in program semantics developed by Cousot-
  +Cousot, Jones, Muchnick, Nielson+Nielson, Hankin, many others.

▶ Research since 1970’s under the name of Abstract Interpretation

▶ Capture a significant part of data flow analysis (but not all).

▶ January 2008 conference in San Francisco:
  “30 Years of Abstract Interpretation.”
Optimising transformations for compilers.

**Compiler structure:**

sourcecode $\rightarrow$ intermediatecode $\rightarrow$ intermediatecode $\rightarrow$ targetcode

**The Optimisation phase:**

intermediatecode $\rightarrow$ intermediatecode

Intermediate code is usually (some version of) simple flow chart programs. These contain

- program points (also called labels),
- with an elementary statement or test at each point, and
- control transitions from one program point to another.
What: program transformation to improve efficiency

► Based on program flow analysis
► Must be correct (and just what does this mean?)
► Complex
► Important: efficiency, complex hardware, limits to what humans can improve, etc

How: several steps in program optimisation. First: program analysis.

► Choose a data flow lattice to describe program properties
► Build a system of data flow equations from the program
► Solve the system of data flow equations

Then transform the program, usually to optimise it
Consider a transformation

\[ [x := a]^\ell \Rightarrow [\text{skip}]^\ell \]

to eliminate code. (It sounds trivial, but it’s significant in practice!)

**Some possible reasons** it can be correct:

1. **Point \( \ell \) is unreachable**: control cannot flow from the program’s start to \([x := a]^\ell\)

2. **Point \( \ell \) is dead**: control cannot flow from \([x := a]^\ell\) to the program’s exit. For example
   - The program will **definitely loop** after point \( \ell \). Or
   - The program will **definitely abort** execution after point \( \ell \).

3. **Variable \( x \) is dead** at \( \ell \) (even though point \( \ell \) is not dead): For instance
   - \( x \) is never referenced again; or
   - \( x \) may be used to compute \( y, z, \ldots \) but they are never used again, . . .
More possible reasons for correctness of the transformation

\[ [x := a]^\ell \Rightarrow [\text{skip}]^\ell \]

to eliminate code.

4. **x** is already equal to **a** (if control ever gets to **\ell**)

5. Mathematical reasons relating **x** and **a**, e.g., Matiyasevich’s theorem etc.

6. **a** is an uninitialised variable: so the value of **x** is completely undependable

7. Some patchwork combination of the above.
   (Eg, reason 3 applies if **x** is even, reason 4 applies if **x** is odd,...)
ALAS, MOST OF THESE REASONS ARE AS UNDECIDABLE AS THE HALTING PROBLEM (!)

Remark: many (most!) of the above program behavior properties are undecidable (if you insist on exact answers).

Proof See Rice’s Theorem from Computability Theory.

So what do we do?

► The practice of program analysis and the theory of abstract interpretation: find safe descriptions of program behavior. Meaning of safety:

• if the analysis says that a program has a certain behavior (e.g., that $x$ is dead at point $\ell$),
• then it definitely has that behavior in all computations.

► However the analysis may be imprecise in this sense:

it can answer “don’t know” even when the property is true.
WHAT KIND OF REASONING CAN BE USED TO DISCOVER PROGRAM PROPERTIES?

They can involve

- **Control flow**, e.g., that point $\ell$ is unreachable
- **Data flow**, e.g., that the value of variable $x$ at point $\ell$ cannot affect the program’s final output.

A useful classification: **dimension 1 = past/future, dimension 2 = may/must.**

- **computational pasts**, e.g., that $x$ equals $a$ if control point $\ell$ is reached
- **computational futures**, e.g., that variable $x$ is dead at control point $\ell$
- **all-path, or “must” properties**, e.g., a past **all-path** property:
  “variable $x$ is initialised”
  i.e., $x$ was set on every computation path from start to current point $\ell$
- **some-path, or “may” properties**, e.g., a future **some-path** property:
  “variable $x$ is live”, i.e., there exists a computation path from current point $\ell$ to the program end
A program analysis will compute a “program-point-centric” analysis that binds information to each program point $\ell$.

The program properties at a program point $\ell$ are

- determined by
  - the computational future (of computations that get as far as $\ell$); or
  - the computational past

- determined by the set of
  - all computation paths from (or to) $\ell$, or by
  - the existence of at least one computation path from (or to) $\ell$
A program analysis will compute a “program-point-centric” analysis that binds information to each program point $\ell$.

Such information (almost always in the literature)

- is finitely (and feasibly!) computable
- is computed uniformly, i.e., for all the source program’s program points.
- Adjacent program points will have properties that are related, e.g., by classic flow equations of dataflow analysis for compiler construction.

An analogy: heat flow equations.

(though heat flows 2-ways, while program flows are asymmetric.)
### SOME NOTATIONS USED IN THE BOOK

<table>
<thead>
<tr>
<th>Symbol</th>
<th>Description</th>
</tr>
</thead>
<tbody>
<tr>
<td>$\ell \in \text{Lab}$</td>
<td>the set of all labels</td>
</tr>
<tr>
<td>$x, y, z \in \text{Var}$</td>
<td>the set of all variables</td>
</tr>
<tr>
<td>$S \in \text{Stmt}$</td>
<td>the set of all statements</td>
</tr>
<tr>
<td>$a \in \text{AExp}$</td>
<td>the set of all arithmetic expressions</td>
</tr>
<tr>
<td>$b \in \text{BExp}$</td>
<td>the set of all Boolean expressions</td>
</tr>
<tr>
<td>$e \in \text{Exp}$</td>
<td>the set of all expressions (arithmetic or Boolean)</td>
</tr>
</tbody>
</table>
ABSTRACT SYNTAX

\[ a ::= x \mid n \mid a_1 \, op \, a_2 \]
\[ b ::= \text{true} \mid \text{false} \mid \text{not} \, b \mid b_1 \, op \, b_2 \mid a_1 \, op \, a_2 \]
\[ S ::= [x := a]^\ell \mid [\text{skip}]^\ell \mid S_1 ; S_2 \]
\[ \quad | \quad \text{if} \ [b]^\ell \ \text{then} \ S_1 \ \text{else} \ S_2 \ | \ \text{while} \ [b]^\ell \ \text{do} \ S \]

\[ B ::= [x := a]^\ell \mid [\text{skip}]^\ell \mid [b]^\ell \]

For our slides: we only think of \text{flow charts containing labeled blocks} \ B, don’t deal with statements that contain other statements. (Doesn’t lose information, and saves notation!)

\textbf{Generic versus concrete:}

\[ [x := a]^\ell \] Math font for \textit{generic} program fragments, e.g.,
\[ x \] ranges over all variables

\[ [x := x+1]^7 \] Teletype font for \textit{concrete} program fragments, e.g.,
the LHS is the concrete variable “x”
A FEW MORE NOTATIONS

\begin{tabular}{|l|}
\hline
\textbf{Lab}_* & the set of all labels \textbf{in the program currently being analysed} \\
\textbf{Var}_* & the set of all variables \textbf{in the program currently being analysed} \\
\textbf{Stmt}_* & the set of all statements \textbf{in the program currently being analysed} \\
\textbf{AExp}_* & the set of all arithmetic expressions \textbf{in the program currently being analysed} \\
\textbf{BExp}_* & the set of all Boolean expressions \textbf{in the program currently being analysed} \\
\hline
\end{tabular}
## 4 USEFUL EXAMPLES OF DATA FLOW ANALYSIS

<table>
<thead>
<tr>
<th>Type of flow equations:</th>
<th>What’s analysed</th>
</tr>
</thead>
<tbody>
<tr>
<td>$F : \text{Lab}_* \rightarrow \text{dataflow lattice } L$</td>
<td>Time dependency</td>
</tr>
<tr>
<td>$RD : \text{Lab}<em>* \rightarrow \mathcal{P}(\text{Var}</em>* \times \text{Lab}_?^*)$</td>
<td>past</td>
</tr>
<tr>
<td>$LV : \text{Lab}<em>* \rightarrow \mathcal{P}(\text{Var}</em>*)$</td>
<td>future</td>
</tr>
<tr>
<td>$AE : \text{Lab}<em>* \rightarrow \mathcal{P}(\text{Exp}</em>*)$</td>
<td>past</td>
</tr>
<tr>
<td>$VB : \text{Lab}<em>* \rightarrow \mathcal{P}(\text{Exp}</em>*)$</td>
<td>future</td>
</tr>
</tbody>
</table>

$RD = \text{Reaching definitions}$ (used for constant propagation)

$LV = \text{Live variables}$ (used for dead code elimination)

$AE = \text{Available expressions}$ (to avoid recomputing expressions)

$VB = \text{Very busy expressions}$ (save expression values for later use)
## INTUITIVE EXPLANATION: LIVE VARIABLES

<table>
<thead>
<tr>
<th>Type of flow equations:</th>
<th>What’s analysed</th>
<th>How it’s computed</th>
</tr>
</thead>
<tbody>
<tr>
<td>$F : \text{Lab}_* \rightarrow \text{dataflow lattice } L$</td>
<td>Time dependency</td>
<td>Path modality</td>
</tr>
<tr>
<td>$LV : \text{Lab}<em>* \rightarrow \mathcal{P}(\text{Var}</em>*)$</td>
<td>future</td>
<td>$\exists$</td>
</tr>
</tbody>
</table>

Variable $x$ is live at program point $\ell$ if there exists a flow chart path from $\ell$ to some usage of variable $x$. Things to notice:

- it’s about what can happen in the future
- along at least one path ($\exists$)

**Optimisation** enabled by live variable analysis:

If $x$ is not live at point $\ell$, then the register / memory cell containing the value of $x$ may be used for another value

Net effect: to reduce memory or register usage.
Expression $e$ is available at program point $\ell$ if on all flow chart paths to $\ell$ the value of $e$ has been computed, and no variable in $e$ has been changed. Things to notice:

- it’s about what did happen in the past
- and along all paths to $\ell$ ($\forall$)

**Optimisation** enabled by live variable analysis:

If $e$ is available at point $\ell$, then (generate code to) fetch the value that has already been computed.

Net effect: generate smaller code.
### INTUITIVE EXPLANATION: VERY BUSY EXPRESSIONS

<table>
<thead>
<tr>
<th>Type of flow equations:</th>
<th>What’s analysed</th>
<th>How it’s computed</th>
</tr>
</thead>
<tbody>
<tr>
<td>$F : \text{Lab}_* \rightarrow \text{dataflow lattice } L$</td>
<td>Time dependency</td>
<td>Data flow</td>
</tr>
<tr>
<td>$V B : \text{Lab}<em>* \rightarrow \mathcal{P}(\text{Exp}</em>*)$</td>
<td>future</td>
<td>Path modality</td>
</tr>
</tbody>
</table>

Expression $e$ is very busy at program point $\ell$ if the value of $e$ will be used on all flow chart paths from $\ell$. Things to notice:

- it’s about what will happen in the future
- and along all paths from $\ell$ ($\forall$)

**Optimisation** enabled by very busy expression analysis:

It can pay to keep the value of $e$ in a register instead of memory.

Net effect: generate faster code.
### INTUITIVE EXPLANATION: REACHING DEFINITIONS

<table>
<thead>
<tr>
<th>Type of flow equations:</th>
<th>What’s analysed</th>
<th>How it’s computed</th>
</tr>
</thead>
<tbody>
<tr>
<td>$F : \text{Lab}_* \rightarrow \text{dataflow lattice } L$</td>
<td>Time dependency, Path modality</td>
<td>Data flow, Kind of fixpoint</td>
</tr>
<tr>
<td>$RD : \text{Lab}<em>* \rightarrow \mathcal{P}(\text{Var}</em>* \times \text{Lab}_*)$</td>
<td>past, $\exists$</td>
<td>forward, least</td>
</tr>
</tbody>
</table>

A pair $(x, \ell_0)$ can reach program point $\ell$ if

- there is a statement $[x := e]^{\ell_0}$, and
- there is a path from $\ell_0$ to $\ell$, and
- variable $x$ is not changed on the path

Things to notice:

- it’s about what happened in the past along at least one path to $\ell$ ($\exists$)

**Optimisation** enabled by reaching definition analysis: **constant propagation**

Net effect: generate faster code.
State: a state is a function $\sigma : \text{Var} \rightarrow \mathbb{Z}$. Also known as a store.

Idea: the current value of variable $x$ is $\sigma(x)$.

A computational configuration is a pair $\langle S, \sigma \rangle$ where $S$ is a statement (what is remaining to execute) and $\sigma$ is the current state.

A one-step transition has form

$$\langle S, \sigma \rangle \rightarrow \langle S', \sigma' \rangle$$

or, if program stops: $\langle S, \sigma \rangle \rightarrow \sigma'$

Details omitted today, but what you would expect. Here there is a data flow from $\sigma$ to $\sigma'$

Each program defines a set of computations. A computation is either

• a terminating computation: a finite sequence

$$\langle S_1, \sigma_1 \rangle \rightarrow \langle S_2, \sigma_2 \rangle \rightarrow \ldots \langle S_n, \sigma_n \rangle \rightarrow \sigma_{n+1}$$

or

• a looping computation: an infinite sequence

$$\langle S_1, \sigma_1 \rangle \rightarrow \langle S_2, \sigma_2 \rangle \rightarrow \ldots$$
THE MAIN PROBLEM OF DFA

Given a program, to find a description of the data flow at each label \( \ell \). In this book, for analysis \( A \):

- \( A_{\text{entry}}(\ell) = \) flow information at the entry to statement \( [B]^{\ell} \)
- \( A_{\text{exit}}(\ell) = \) flow information at the exit from statement \( [B]^{\ell} \)

Suppose program has the form:

\[
[B_1]^{\ell_1} [B_2]^{\ell_2} \ldots [B_n]^{\ell_n}
\]

Then a program description will have the form:

\[
A_{\text{entry}} : \text{Lab}_* \rightarrow L \quad \text{and} \quad A_{\text{exit}} : \text{Lab}_* \rightarrow L
\]

where \( L \) is a complete lattice. Different lattices for different flow properties.

**Flow lattice:** a structure \( L = (L, \sqsubseteq, \cup, \cap, \bot, \top) \).
Program:

\[ [y := x]^1; [z := 1]^2; \text{while } [y > 1]^3 \text{ do } ([z := z \times y]^4; [y := y - 1]^5); [y := 0]^6; \]

**Reaching definitions lattice:**

\[ L = ( \mathcal{P}(\{x, y, z\} \times \{1, 2, 3, 4, 5, 6, ?\}), \subseteq, \sqcup, \sqcap, \perp, \top) \]

\((x, \ell_0) \in RD_\downarrow(\ell)\) if for some computation path from \(\ell_0\) to \(\ell\)

\\(|x\) was assigned at point \(\ell_0\), and (jargon: “defined”)\\(|x\) was not re-assigned before point \(\ell\) (i.e., the assignment “reaches” \(\ell\))

**Uninitialised variables:** are “reached” from point “?”

<table>
<thead>
<tr>
<th>(\ell)</th>
<th>(RD_{entry}(\ell))</th>
<th>(RD_{exit}(\ell))</th>
</tr>
</thead>
<tbody>
<tr>
<td>1</td>
<td>{(x,?), (y,?), (z,?)}</td>
<td>{(x,?), (y, 1), (z,?)}</td>
</tr>
<tr>
<td>2</td>
<td>{(x,?), (y, 1), (z,?)}</td>
<td>{(x,?), (y, 1), (z,2)}</td>
</tr>
<tr>
<td>3</td>
<td>{(x,?), (y, 1), (y, 5), (z,2), (z,4)}</td>
<td>{(x,?), (y, 1), (y, 5), (z,2), (z,4)}</td>
</tr>
<tr>
<td>4</td>
<td>{(x,?), (y, 1), (y, 5), (z,2), (z,4)}</td>
<td>{(x,?), (y, 1), (y, 5), (z,4)}</td>
</tr>
<tr>
<td>5</td>
<td>{(x,?), (y, 1), (y, 5), (z,4)}</td>
<td>{(x,?), (y, 5), (z,4)}</td>
</tr>
<tr>
<td>6</td>
<td>{(x,?), (y, 1), (y, 5), (z,2), (z,4)}</td>
<td>{(x,?), (y, 6), (z,2), (z,4)}</td>
</tr>
</tbody>
</table>
A FUTURE ANALYSIS: LIVE VARIABLES  
(for the same program to compute $x!$)

Program:

$[y:=x]^1;[z:=1]^2; \text{while } [y>1]^3 \text{ do } ([z:=z*y]^4;[y:=y-1]^5); [y:=0]^6;$

Live variable lattice:

$$L = (\mathcal{P}(\{x, y, z\}), \subseteq, \cup, \cap, \bot, \top)$$

Variable $x$ is live if $\exists$ computation path with a future reference to $x$.

Assume: no variables are live at program exit.

<table>
<thead>
<tr>
<th>$\ell$</th>
<th>$LV_{entry}(\ell)$</th>
<th>$LV_{exit}(\ell)$</th>
</tr>
</thead>
<tbody>
<tr>
<td>1</td>
<td>{x}</td>
<td>{y}</td>
</tr>
<tr>
<td>2</td>
<td>{y}</td>
<td>{y, z}</td>
</tr>
<tr>
<td>3</td>
<td>{y, z}</td>
<td>{y, z}</td>
</tr>
<tr>
<td>4</td>
<td>{y, z}</td>
<td>{y, z}</td>
</tr>
<tr>
<td>5</td>
<td>{y, z}</td>
<td>{y, z}</td>
</tr>
<tr>
<td>6</td>
<td>$\emptyset$</td>
<td>$\emptyset$</td>
</tr>
</tbody>
</table>
LIVE VARIABLE FLOW EQUATIONS:
(for the same program to compute $x!$)

$[y:=x]^1;[z:=1]^2; \text{while } [y>1]^3 \text{ do } ([z:=z*y]^4;[y:=y-1]^5);[y:=0]^6;$

\[
LV_{entry}(1) = LV_{exit}(1) \setminus \{y\} \cup \{x\} \\
LV_{entry}(2) = LV_{exit}(2) \setminus \{z\} \\
LV_{entry}(3) = LV_{exit}(3) \cup \{y\} \\
LV_{entry}(4) = LV_{exit}(4) \cup \{y, z\} \\
LV_{entry}(5) = LV_{exit}(5) \cup \{y\} \\
LV_{entry}(6) = LV_{exit}(6) \setminus \{y\} \\
LV_{exit}(1) = LV_{entry}(2) \\
LV_{exit}(2) = LV_{entry}(3) \\
LV_{exit}(3) = LV_{entry}(4) \cup LV_{entry}(6) \\
LV_{exit}(4) = LV_{entry}(5) \\
LV_{exit}(5) = LV_{entry}(3) \\
LV_{exit}(6) = \emptyset
\]
WHAT ON EARTH IS GOING ON?

▶ What is being defined by these equations?
▶ What data flow logic is being expressed?
▶ How can the equations be solved?

The equations define the values of in all 12 program point descriptions

\[ LV_{entry}(1), \ldots, LV_{entry}(6), LV_{exit}(1), \ldots, LV_{exit}(6) \]

in terms of each other.

This is a recursive system of data flow equations to describe the program’s computational behavior.

Solution to the equation system: This is called a fixpoint.

Type of a solution to the equation system: \( L^{12} \), where \( L \) is the description data flow lattice.

Type of the equation system itself:

\[ F : L^{12} \rightarrow L^{12} \]
**Time dependence.** Possibilities:

- **Future analysis:** the property depends on the *computational future*. Computed by *backward data flow*.

- **Past analysis:** the property depends on the *computational past*. Computed by *forward data flow*. – “must” or “may” dependence:

**Path modality dependence.** Possibilities:

- **may** path dependence (for some path)
- **must** path dependence (for all paths)

These make *four combinations*. For example:

- Both $LV$ and $RD$ are **may** path dependencies
- Live variables $LV$ is a **future** analysis (= backward data flow)
- Reaching definitions $RD$ is a **past** analysis (= forward data flow)
FLOW EQUATIONS: REFLECT THE 4 COMBINATIONS

Future/past: what is defined in terms of what in the equations, e.g.,

future: $LV_{entry}(\ell) = \ldots LV_{exit}(\ell) \ldots$
past: $LV_{exit}(\ell) = \ldots LV_{entry}(\ell) \ldots$

All-paths/some-path: find greatest or least fixpoint solution to equations

Fixpoints: $lfp$ (least fixpoint) for $\exists$ path dependence

$$lfp(F) = \bigsqcup_{n \to \infty} F^n(\bot, \bot, \ldots, \bot)$$

$gfp$ (greatest fixpoint) for $\forall$ path dependence

$$gfp(F) = \bigcap_{n \to \infty} F^n(\top, \top, \ldots, \top)$$

Combining flows from several blocks into one:

- Use $\sqcup$ when computing least fixpoint (some-path properties)
- Use $\sqcap$ when computing greatest fixpoint (all-path properties)
4 EXAMPLES OF THE 4 COMBINATIONS

<table>
<thead>
<tr>
<th>Type of flow equations:</th>
<th>What’s analysed</th>
<th>How it’s computed</th>
</tr>
</thead>
<tbody>
<tr>
<td>$F : \text{Lab}_* \rightarrow \text{dataflow lattice } L$</td>
<td>Time dependency</td>
<td>Path modality</td>
</tr>
<tr>
<td>$RD : \text{Lab}<em>* \rightarrow \mathcal{P}(\text{Var}</em>* \times \text{Lab}_?^*)$</td>
<td>past</td>
<td>$\exists$</td>
</tr>
<tr>
<td>$LV : \text{Lab}<em>* \rightarrow \mathcal{P}(\text{Var}</em>*)$</td>
<td>future</td>
<td>$\exists$</td>
</tr>
<tr>
<td>$AE : \text{Lab}<em>* \rightarrow \mathcal{P}(\text{Exp}</em>*)$</td>
<td>past</td>
<td>$\forall$</td>
</tr>
<tr>
<td>$VB : \text{Lab}<em>* \rightarrow \mathcal{P}(\text{Exp}</em>*)$</td>
<td>future</td>
<td>$\forall$</td>
</tr>
</tbody>
</table>

$RD =$ Reaching definitions  
$LV =$ Live variables  
$AE =$ Available expressions  
$VB =$ Very busy expressions
Form of the data flow equation system:

\[ (X_1, X_2, \ldots, X_{2n}) = (e_1(\bar{X}), e_2(\bar{X}), \ldots, e_{2n}(\bar{X})) \]

where set expressions \( e_1, \ldots, e_{2n} \) are built from \( X_1, X_2, \ldots, X_{2n} \) by set operations such as \( \cup, \cap, \setminus \) and constants.

This defines a function

\[ F : \mathcal{P}(D)^{2n} \to \mathcal{P}(D)^{2n} \]

(where \( D = \) set of descriptions, \( n = \) number of labels)

on the lattice

\[ L = (L, \subseteq, \cup, \cap, \bot, \top) = (\mathcal{P}(D), \subseteq, \cup, \cap, \emptyset, D) \]

**Fixpoints:**

\[ \text{lfp}(F) = \bigcup_{n \to \infty} F^n(\bot, \bot, \ldots, \bot), \quad \text{gfp}(F) = \bigcap_{n \to \infty} F^n(\top, \ldots, \top) \]

1. \( \mathcal{P}(D) \) is a lattice, so \( F(X_1, X_2, \ldots, X_{2n}) \) exists.
2. \( \mathcal{P}(D) \) is complete, so \( \text{lfp}(F), \text{gfp}(F) \) both exist.
3. Ascending (descending) chain condition: ensures that

\[ \text{lfp}(F), \text{gfp}(F) \] are finitely computable.
CHAOTIC ITERATION

Effect is to compute the least (or greatest) fixpoint by repeatedly applying the equations

- Apply them in any order
- until no sets can be changed

- Initialisation of the sets:
  - Least fixpoint: start with every set empty (⊥ of the lattice)
  - Greatest fixpoint: start with every set equal to (⊤ of the lattice)

- Amazing fact: it doesn’t matter what order is chosen (hence the name “chaotic”)
HAND SOLUTION: LIVE VARIABLE ANALYSIS OF THE FACTORIAL PROGRAM BY CHAOTIC ITERATION 1

$[y:=x]; [z:=1]; \text{while } [y>1] \text{ do } ([z:=z*y]; [y:=y-1]); [y:=0];$

$LV_{entry}(1) = LV_{exit}(1) \setminus \{y\} \cup \{x\}$

$LV_{entry}(2) = LV_{exit}(2) \setminus \{z\}$

$LV_{entry}(3) = LV_{exit}(3) \cup \{y\}$

$LV_{entry}(4) = LV_{exit}(4) \cup \{y, z\}$

$LV_{entry}(5) = LV_{exit}(5) \cup \{y\}$

$LV_{entry}(6) = LV_{exit}(6) \setminus \{y\}$

$LV_{exit}(1) = LV_{entry}(2)$

$LV_{exit}(2) = LV_{entry}(3)$

$LV_{exit}(3) = LV_{entry}(4) \cup LV_{entry}(6)$

$LV_{exit}(4) = LV_{entry}(5)$

$LV_{exit}(5) = LV_{entry}(3)$

$LV_{exit}(6) = \emptyset$

<table>
<thead>
<tr>
<th>$\ell$</th>
<th>$LV_{entry}(\ell)$</th>
<th>$LV_{exit}(\ell)$</th>
</tr>
</thead>
<tbody>
<tr>
<td>1</td>
<td>$\emptyset$</td>
<td>$\emptyset$</td>
</tr>
<tr>
<td>2</td>
<td>$\emptyset$</td>
<td>$\emptyset$</td>
</tr>
<tr>
<td>3</td>
<td>$\emptyset$</td>
<td>$\emptyset$</td>
</tr>
<tr>
<td>4</td>
<td>$\emptyset$</td>
<td>$\emptyset$</td>
</tr>
<tr>
<td>5</td>
<td>$\emptyset$</td>
<td>$\emptyset$</td>
</tr>
<tr>
<td>6</td>
<td>$\emptyset$</td>
<td>$\emptyset$</td>
</tr>
</tbody>
</table>
HAND SOLUTION: LIVE VARIABLE ANALYSIS OF THE FACTORIAL PROGRAM BY CHAOTIC ITERATION 2

\[ y := x; \quad z := 1; \quad \text{while} \quad [y > 1] \quad \text{do} \quad ([z := z \times y]; [y := y - 1]); [y := 0]; \]

\[
\begin{align*}
LV_{entry}(1) & = LV_{exit}(1) \setminus \{y\} \cup \{x\} \\
LV_{entry}(2) & = LV_{exit}(2) \setminus \{z\} \\
LV_{entry}(3) & = LV_{exit}(3) \cup \{y\} \\
LV_{entry}(4) & = LV_{exit}(4) \cup \{y, z\} \\
LV_{entry}(5) & = LV_{exit}(5) \cup \{y\} \\
LV_{entry}(6) & = LV_{exit}(6) \setminus \{y\} \\
LV_{exit}(1) & = LV_{entry}(2) \\
LV_{exit}(2) & = LV_{entry}(3) \\
LV_{exit}(3) & = LV_{entry}(4) \cup LV_{entry}(6) \\
LV_{exit}(4) & = LV_{entry}(5) \\
LV_{exit}(5) & = LV_{entry}(3) \\
LV_{exit}(6) & = \emptyset
\end{align*}
\]

<table>
<thead>
<tr>
<th>( \ell )</th>
<th>( LV_{entry}(\ell) )</th>
<th>( LV_{exit}(\ell) )</th>
</tr>
</thead>
<tbody>
<tr>
<td>1</td>
<td>{x}</td>
<td>\emptyset</td>
</tr>
<tr>
<td>2</td>
<td>\emptyset</td>
<td>\emptyset</td>
</tr>
<tr>
<td>3</td>
<td>{y}</td>
<td>\emptyset</td>
</tr>
<tr>
<td>4</td>
<td>{y, z}</td>
<td>\emptyset</td>
</tr>
<tr>
<td>5</td>
<td>{y}</td>
<td>\emptyset</td>
</tr>
<tr>
<td>6</td>
<td>\emptyset</td>
<td>\emptyset</td>
</tr>
</tbody>
</table>
[y:=x]$^1$;[z:=1]$^2$; while [y>1]$^3$ do ([z:=z*y]$^4$;[y:=y-1]$^5$);[y:=0]$^6$;

\[
\begin{align*}
L_{\text{entry}}(1) &= L_{\text{exit}}(1) \setminus \{y\} \cup \{x\} \\
L_{\text{entry}}(2) &= L_{\text{exit}}(2) \setminus \{z\} \\
L_{\text{entry}}(3) &= L_{\text{exit}}(3) \cup \{y\} \\
L_{\text{entry}}(4) &= L_{\text{exit}}(4) \cup \{y, z\} \\
L_{\text{entry}}(5) &= L_{\text{exit}}(5) \cup \{y\} \\
L_{\text{entry}}(6) &= L_{\text{exit}}(6) \setminus \{y\} \\
L_{\text{exit}}(1) &= L_{\text{entry}}(2) \\
L_{\text{exit}}(2) &= L_{\text{entry}}(3) \\
L_{\text{exit}}(3) &= L_{\text{entry}}(4) \cup L_{\text{entry}}(6) \\
L_{\text{exit}}(4) &= L_{\text{entry}}(5) \\
L_{\text{exit}}(5) &= L_{\text{entry}}(3) \\
L_{\text{exit}}(6) &= \emptyset
\end{align*}
\]

<table>
<thead>
<tr>
<th>$\ell$</th>
<th>$L_{\text{entry}}(\ell)$</th>
<th>$L_{\text{exit}}(\ell)$</th>
</tr>
</thead>
<tbody>
<tr>
<td>1</td>
<td>${x}$</td>
<td>$\emptyset$</td>
</tr>
<tr>
<td>2</td>
<td>$\emptyset$</td>
<td>${y}$</td>
</tr>
<tr>
<td>3</td>
<td>${y}$</td>
<td>$\emptyset$</td>
</tr>
<tr>
<td>4</td>
<td>${y, z}$</td>
<td>$\emptyset$</td>
</tr>
<tr>
<td>5</td>
<td>${y}$</td>
<td>$\emptyset$</td>
</tr>
<tr>
<td>6</td>
<td>$\emptyset$</td>
<td>$\emptyset$</td>
</tr>
</tbody>
</table>
HAND SOLUTION: LIVE VARIABLE ANALYSIS OF THE FACTORIAL PROGRAM BY CHAOTIC ITERATION 4

\[ y := 1; [z := 1] \text{ while } [y > 1] \text{ do } ([z := z \times y]; [y := y - 1]); [y := 0]; \]

\[
\begin{align*}
LV_{\text{entry}}(1) & = LV_{\text{exit}}(1) \setminus \{y\} \cup \{x\} \\
LV_{\text{entry}}(2) & = LV_{\text{exit}}(2) \setminus \{z\} \\
LV_{\text{entry}}(3) & = LV_{\text{exit}}(3) \cup \{y\} \\
LV_{\text{entry}}(4) & = LV_{\text{exit}}(4) \cup \{y, z\} \\
LV_{\text{entry}}(5) & = LV_{\text{exit}}(5) \cup \{y\} \\
LV_{\text{entry}}(6) & = LV_{\text{exit}}(6) \setminus \{y\} \\
LV_{\text{exit}}(1) & = LV_{\text{entry}}(2) \\
LV_{\text{exit}}(2) & = LV_{\text{entry}}(3) \\
LV_{\text{exit}}(3) & = LV_{\text{entry}}(4) \cup LV_{\text{entry}}(6) \\
LV_{\text{exit}}(4) & = LV_{\text{entry}}(5) \\
LV_{\text{exit}}(5) & = LV_{\text{entry}}(3) \\
LV_{\text{exit}}(6) & = \emptyset
\end{align*}
\]

<table>
<thead>
<tr>
<th>( \ell )</th>
<th>( LV_{\text{entry}}(\ell) )</th>
<th>( LV_{\text{exit}}(\ell) )</th>
</tr>
</thead>
<tbody>
<tr>
<td>1</td>
<td>{x}</td>
<td>\emptyset</td>
</tr>
<tr>
<td>2</td>
<td>\emptyset</td>
<td>{y}</td>
</tr>
<tr>
<td>3</td>
<td>{y}</td>
<td>{y, z}</td>
</tr>
<tr>
<td>4</td>
<td>{y, z}</td>
<td>\emptyset</td>
</tr>
<tr>
<td>5</td>
<td>{y}</td>
<td>\emptyset</td>
</tr>
<tr>
<td>6</td>
<td>\emptyset</td>
<td>\emptyset</td>
</tr>
</tbody>
</table>
HAND SOLUTION: LIVE VARIABLE ANALYSIS OF THE FACTORIAL PROGRAM BY CHAOTIC ITERATION 5

\[y := x^1; [z := 1]^2; \text{while } [y > 1]^3 \text{ do } ([z := z \times y]^4; [y := y - 1]^5); [y := 0]^6;\]

\[
\begin{align*}
LV_{\text{entry}}(1) &= LV_{\text{exit}}(1) \setminus \{y\} \cup \{x\} \\
LV_{\text{entry}}(2) &= LV_{\text{exit}}(2) \setminus \{z\} \\
LV_{\text{entry}}(3) &= LV_{\text{exit}}(3) \cup \{y\} \\
LV_{\text{entry}}(4) &= LV_{\text{exit}}(4) \cup \{y, z\} \\
LV_{\text{entry}}(5) &= LV_{\text{exit}}(5) \cup \{y\} \\
LV_{\text{entry}}(6) &= LV_{\text{exit}}(6) \setminus \{y\} \\
LV_{\text{exit}}(1) &= LV_{\text{entry}}(2) \\
LV_{\text{exit}}(2) &= LV_{\text{entry}}(3) \\
LV_{\text{exit}}(3) &= LV_{\text{entry}}(4) \cup LV_{\text{entry}}(6) \\
LV_{\text{exit}}(4) &= LV_{\text{entry}}(5) \\
LV_{\text{exit}}(5) &= LV_{\text{entry}}(3) \\
LV_{\text{exit}}(6) &= \emptyset
\end{align*}
\]

\[
\begin{array}{|c|c|c|}
\hline
\ell & \text{LV}_{\text{entry}}(\ell) & \text{LV}_{\text{exit}}(\ell) \\
\hline
1 & \{x\} & \emptyset \\
2 & \emptyset & \{y\} \\
3 & \{y\} & \{y, z\} \\
4 & \{y, z\} & \{y\} \\
5 & \{y\} & \emptyset \\
6 & \emptyset & \emptyset \\
\hline
\end{array}
\]
[y:=x]¹; [z:=1]²; while [y>1]³ do ([z:=z*y]⁴; [y:=y-1]⁵); [y:=0]⁶;

\[
\begin{align*}
LV_{\text{entry}}(1) &= LV_{\text{exit}}(1) \setminus \{y\} \cup \{x\} \\
LV_{\text{entry}}(2) &= LV_{\text{exit}}(2) \setminus \{z\} \\
LV_{\text{entry}}(3) &= LV_{\text{exit}}(3) \cup \{y\} \\
LV_{\text{entry}}(4) &= LV_{\text{exit}}(4) \cup \{y, z\} \\
LV_{\text{entry}}(5) &= LV_{\text{exit}}(5) \cup \{y\} \\
LV_{\text{entry}}(6) &= LV_{\text{exit}}(6) \setminus \{y\} \\
LV_{\text{exit}}(1) &= LV_{\text{entry}}(2) \\
LV_{\text{exit}}(2) &= LV_{\text{entry}}(3) \\
LV_{\text{exit}}(3) &= LV_{\text{entry}}(4) \cup LV_{\text{entry}}(6) \\
LV_{\text{exit}}(4) &= LV_{\text{entry}}(5) \\
LV_{\text{exit}}(5) &= LV_{\text{entry}}(3) \\
LV_{\text{exit}}(6) &= \emptyset
\end{align*}
\]

\[
\begin{array}{|c|c|c|}
\hline
\ell & LV_{\text{entry}}(\ell) & LV_{\text{exit}}(\ell) \\
\hline
1 & \{x\} & \emptyset \\
2 & \emptyset & \{y\} \\
3 & \{y\} & \{y, z\} \\
4 & \{y, z\} & \{y\} \\
5 & \{y\} & \{y\} \\
6 & \emptyset & \emptyset \\
\hline
\end{array}
\]
HAND SOLUTION: LIVE VARIABLE ANALYSIS OF THE FACTORIAL PROGRAM BY CHAOTIC ITERATION 7

\[ y := x^1; [z := 1]^2; \text{while } [y > 1]^3 \text{ do } ([z := z \times y]^4; [y := y - 1]^5); [y := 0]^6; \]

\[
\begin{align*}
LV_{\text{entry}}(1) &= LV_{\text{exit}}(1) \setminus \{y\} \cup \{x\} \\
LV_{\text{entry}}(2) &= LV_{\text{exit}}(2) \setminus \{z\} \\
LV_{\text{entry}}(3) &= LV_{\text{exit}}(3) \cup \{y\} \\
LV_{\text{entry}}(4) &= LV_{\text{exit}}(4) \cup \{y, z\} \\
LV_{\text{entry}}(5) &= LV_{\text{exit}}(5) \cup \{y\} \\
LV_{\text{entry}}(6) &= LV_{\text{exit}}(6) \setminus \{y\} \\
LV_{\text{exit}}(1) &= LV_{\text{entry}}(2) \\
LV_{\text{exit}}(2) &= LV_{\text{entry}}(3) \\
LV_{\text{exit}}(3) &= LV_{\text{entry}}(4) \cup LV_{\text{entry}}(6) \\
LV_{\text{exit}}(4) &= LV_{\text{entry}}(5) \\
LV_{\text{exit}}(5) &= LV_{\text{entry}}(3) \\
LV_{\text{exit}}(6) &= \emptyset
\end{align*}
\]

<table>
<thead>
<tr>
<th>\ell</th>
<th>LV_{\text{entry}}(\ell)</th>
<th>LV_{\text{exit}}(\ell)</th>
</tr>
</thead>
<tbody>
<tr>
<td>1</td>
<td>{x}</td>
<td>\emptyset</td>
</tr>
<tr>
<td>2</td>
<td>{y}</td>
<td>{y}</td>
</tr>
<tr>
<td>3</td>
<td>{y}</td>
<td>{y, z}</td>
</tr>
<tr>
<td>4</td>
<td>{y, z}</td>
<td>{y}</td>
</tr>
<tr>
<td>5</td>
<td>{y}</td>
<td>{y}</td>
</tr>
<tr>
<td>6</td>
<td>\emptyset</td>
<td>\emptyset</td>
</tr>
</tbody>
</table>

— 37 —
[y:=x]$^1$;[z:=1]$^2$; while [y>1]$^3$ do ([z:=z*y]$^4$;[y:=y-1]$^5$);[y:=0]$^6$;

\[
\begin{align*}
LV_{\text{entry}}(1) &= LV_{\text{exit}}(1) \setminus \{y\} \cup \{x\} \\
LV_{\text{entry}}(2) &= LV_{\text{exit}}(2) \setminus \{z\} \\
LV_{\text{entry}}(3) &= LV_{\text{exit}}(3) \cup \{y\} \\
LV_{\text{entry}}(4) &= LV_{\text{exit}}(4) \cup \{y, z\} \\
LV_{\text{entry}}(5) &= LV_{\text{exit}}(5) \cup \{y, z\} \\
LV_{\text{entry}}(6) &= LV_{\text{exit}}(6) \setminus \{y\} \\
LV_{\text{exit}}(1) &= LV_{\text{entry}}(2) \\
LV_{\text{exit}}(2) &= LV_{\text{entry}}(3) \\
LV_{\text{exit}}(3) &= LV_{\text{entry}}(4) \cup LV_{\text{entry}}(6) \\
LV_{\text{exit}}(4) &= LV_{\text{entry}}(5) \\
LV_{\text{exit}}(5) &= LV_{\text{entry}}(3) \\
LV_{\text{exit}}(6) &= \emptyset \\
\end{align*}
\]

<table>
<thead>
<tr>
<th>(\ell)</th>
<th>(LV_{\text{entry}}(\ell))</th>
<th>(LV_{\text{exit}}(\ell))</th>
</tr>
</thead>
<tbody>
<tr>
<td>1</td>
<td>{x}</td>
<td>\emptyset</td>
</tr>
<tr>
<td>2</td>
<td>{y}</td>
<td>{y}</td>
</tr>
<tr>
<td>3</td>
<td>{y, z}</td>
<td>{y, z}</td>
</tr>
<tr>
<td>4</td>
<td>{y, z}</td>
<td>{y}</td>
</tr>
<tr>
<td>5</td>
<td>{y}</td>
<td>{y}</td>
</tr>
<tr>
<td>6</td>
<td>\emptyset</td>
<td>\emptyset</td>
</tr>
</tbody>
</table>
[y:=x]^{1};[z:=1]^{2};\text{while } [y>1]^{3} \text{ do } ([z:=z*y]^{4};[y:=y-1]^{5});[y:=0]^{6};$

\begin{align*}
LV_{\text{entry}}(1) &= LV_{\text{exit}}(1) \setminus \{y\} \cup \{x\} \\
LV_{\text{entry}}(2) &= LV_{\text{exit}}(2) \setminus \{z\} \\
LV_{\text{entry}}(3) &= LV_{\text{exit}}(3) \cup \{y\} \\
LV_{\text{entry}}(4) &= LV_{\text{exit}}(4) \cup \{y, z\} \\
LV_{\text{entry}}(5) &= LV_{\text{exit}}(5) \cup \{y\} \\
LV_{\text{entry}}(6) &= LV_{\text{exit}}(6) \setminus \{y\} \\
LV_{\text{exit}}(1) &= LV_{\text{entry}}(2) \\
LV_{\text{exit}}(2) &= LV_{\text{entry}}(3) \\
LV_{\text{exit}}(3) &= LV_{\text{entry}}(4) \cup LV_{\text{entry}}(6) \\
LV_{\text{exit}}(4) &= LV_{\text{entry}}(5) \\
LV_{\text{exit}}(5) &= LV_{\text{entry}}(3) \\
LV_{\text{exit}}(6) &= \emptyset
\end{align*}$

<table>
<thead>
<tr>
<th>$\ell$</th>
<th>$LV_{\text{entry}}(\ell)$</th>
<th>$LV_{\text{exit}}(\ell)$</th>
</tr>
</thead>
<tbody>
<tr>
<td>1</td>
<td>${x}$</td>
<td>${y}$</td>
</tr>
<tr>
<td>2</td>
<td>${y}$</td>
<td>${y}$</td>
</tr>
<tr>
<td>3</td>
<td>${y, z}$</td>
<td>${y, z}$</td>
</tr>
<tr>
<td>4</td>
<td>${y, z}$</td>
<td>${y}$</td>
</tr>
<tr>
<td>5</td>
<td>${y}$</td>
<td>${y}$</td>
</tr>
<tr>
<td>6</td>
<td>$\emptyset$</td>
<td>$\emptyset$</td>
</tr>
</tbody>
</table>

LV_{entry}(1) = LV_{exit}(1) \{y\} \cup \{x\}
LV_{entry}(2) = LV_{exit}(2) \{z\}
LV_{entry}(3) = LV_{exit}(3) \cup \{y\}
LV_{entry}(4) = LV_{exit}(4) \cup \{y, z\}
LV_{entry}(5) = LV_{exit}(5) \cup \{y\}
LV_{entry}(6) = LV_{exit}(6) \{y\}
LV_{exit}(1) = LV_{entry}(2)
LV_{exit}(2) = LV_{entry}(3)
LV_{exit}(3) = LV_{entry}(4) \cup LV_{entry}(6)
LV_{exit}(4) = LV_{entry}(5)
LV_{exit}(5) = LV_{entry}(3)
LV_{exit}(6) = \emptyset

<table>
<thead>
<tr>
<th>\ell</th>
<th>LV_{entry}(\ell)</th>
<th>LV_{exit}(\ell)</th>
</tr>
</thead>
<tbody>
<tr>
<td>1</td>
<td>{x}</td>
<td>{y}</td>
</tr>
<tr>
<td>2</td>
<td>{y}</td>
<td>{y, z}</td>
</tr>
<tr>
<td>3</td>
<td>{y, z}</td>
<td>{y, z}</td>
</tr>
<tr>
<td>4</td>
<td>{y, z}</td>
<td>{y}</td>
</tr>
<tr>
<td>5</td>
<td>{y}</td>
<td>{y}</td>
</tr>
<tr>
<td>6</td>
<td>\emptyset</td>
<td>\emptyset</td>
</tr>
</tbody>
</table>

\[
\begin{align*}
LV_{entry}(1) & = LV_{exit}(1) \setminus \{y\} \cup \{x\} \\
LV_{entry}(2) & = LV_{exit}(2) \setminus \{z\} \\
LV_{entry}(3) & = LV_{exit}(3) \cup \{y\} \\
LV_{entry}(4) & = LV_{exit}(4) \cup \{y,z\} \\
LV_{entry}(5) & = LV_{exit}(5) \cup \{y\} \\
LV_{entry}(6) & = LV_{exit}(6) \setminus \{y\} \\
LV_{exit}(1) & = LV_{entry}(2) \\
LV_{exit}(2) & = LV_{entry}(3) \\
LV_{exit}(3) & = LV_{entry}(4) \cup LV_{entry}(6) \\
LV_{exit}(4) & = LV_{entry}(5) \\
LV_{exit}(5) & = LV_{entry}(3) \\
LV_{exit}(6) & = \emptyset
\end{align*}
\]

<table>
<thead>
<tr>
<th>ℓ</th>
<th>(LV_{entry}(ℓ))</th>
<th>(LV_{exit}(ℓ))</th>
</tr>
</thead>
<tbody>
<tr>
<td>1</td>
<td>{x}</td>
<td>{y}</td>
</tr>
<tr>
<td>2</td>
<td>{y}</td>
<td>{y,z}</td>
</tr>
<tr>
<td>3</td>
<td>{y,z}</td>
<td>{y,z}</td>
</tr>
<tr>
<td>4</td>
<td>{y,z}</td>
<td>{y}</td>
</tr>
<tr>
<td>5</td>
<td>{y}</td>
<td>{y,z}</td>
</tr>
<tr>
<td>6</td>
<td>\emptyset</td>
<td>\emptyset</td>
</tr>
</tbody>
</table>
HAND SOLUTION: LIVE VARIABLE ANALYSIS OF THE FACTORIAL PROGRAM BY CHAOTIC ITERATION 12

\[ y := x \] \( ^1 \); [z := 1] \( ^2 \); while [y > 1] \( ^3 \) do ([z := z*y] \( ^4 \); [y := y - 1] \( ^5 \)); [y := 0] \( ^6 \);

\[
\begin{align*}
 LV_{entry}(1) &= LV_{exit}(1) \setminus \{y\} \cup \{x\} \\
 LV_{entry}(2) &= LV_{exit}(2) \setminus \{z\} \\
 LV_{entry}(3) &= LV_{exit}(3) \cup \{y\} \\
 LV_{entry}(4) &= LV_{exit}(4) \cup \{y, z\} \\
 LV_{entry}(5) &= LV_{exit}(5) \cup \{y\} \\
 LV_{entry}(6) &= LV_{exit}(6) \setminus \{y\} \\
 LV_{exit}(1) &= LV_{entry}(2) \\
 LV_{exit}(2) &= LV_{entry}(3) \\
 LV_{exit}(3) &= LV_{entry}(4) \cup LV_{entry}(6) \\
 LV_{exit}(4) &= LV_{entry}(5) \\
 LV_{exit}(5) &= LV_{entry}(3) \\
 LV_{exit}(6) &= \emptyset
\end{align*}
\]

<table>
<thead>
<tr>
<th>( \ell )</th>
<th>( LV_{entry}(\ell) )</th>
<th>( LV_{exit}(\ell) )</th>
</tr>
</thead>
<tbody>
<tr>
<td>1</td>
<td>{x}</td>
<td>{y}</td>
</tr>
<tr>
<td>2</td>
<td>{y}</td>
<td>{y, z}</td>
</tr>
<tr>
<td>3</td>
<td>{y, z}</td>
<td>{y, z}</td>
</tr>
<tr>
<td>4</td>
<td>{y, z}</td>
<td>{y}</td>
</tr>
<tr>
<td>5</td>
<td>{y, z}</td>
<td>{y, z}</td>
</tr>
<tr>
<td>6</td>
<td>\emptyset</td>
<td>\emptyset</td>
</tr>
</tbody>
</table>
HAND SOLUTION: LIVE VARIABLE ANALYSIS OF THE FACTORIAL PROGRAM BY CHAOTIC ITERATION 13

\[ y := x \]
\[ z := 1 \] while \[ y > 1 \] do ( \[ z := z \times y \]; \[ y := y - 1 \] ) ; \[ y := 0 \];

\[
\begin{align*}
LV_{\text{entry}}(1) &= LV_{\text{exit}}(1) \setminus \{y\} \cup \{x\} \\
LV_{\text{entry}}(2) &= LV_{\text{exit}}(2) \setminus \{z\} \\
LV_{\text{entry}}(3) &= LV_{\text{exit}}(3) \cup \{y\} \\
LV_{\text{entry}}(4) &= LV_{\text{exit}}(4) \cup \{y, z\} \\
LV_{\text{entry}}(5) &= LV_{\text{exit}}(5) \cup \{y\} \\
LV_{\text{entry}}(6) &= LV_{\text{exit}}(6) \setminus \{y\} \\
LV_{\text{exit}}(1) &= LV_{\text{entry}}(2) \\
LV_{\text{exit}}(2) &= LV_{\text{entry}}(3) \\
LV_{\text{exit}}(3) &= LV_{\text{entry}}(4) \cup LV_{\text{entry}}(6) \\
LV_{\text{exit}}(4) &= LV_{\text{entry}}(5) \\
LV_{\text{exit}}(5) &= LV_{\text{entry}}(3) \\
LV_{\text{exit}}(6) &= \emptyset
\end{align*}
\]

\[
\begin{array}{|c|c|c|}
\hline
\ell & LV_{\text{entry}}(\ell) & LV_{\text{exit}}(\ell) \\
\hline
1 & \{x\} & \{y\} \\
2 & \{y\} & \{y, z\} \\
3 & \{y, z\} & \{y, z\} \\
4 & \{y, z\} & \{y, z\} \\
5 & \{y, z\} & \{y, z\} \\
6 & \emptyset & \emptyset \\
\hline
\end{array}
\]
THE ITERATION PROCESS CONVERGED! (AT LAST)

Chaotic iteration:

► this always works, i.e., it always converges and to the same fixpoint

► the final result is a safe description of the program’s data flow.

► some iteration orders converge faster than others.
Short answer: the result of much experience in writing analysis phases for real compilers. We’ll see some examples.

Future/past: what is defined in terms of what in the equations, e.g.,

\[ LV_{\text{entry}}(\ell) = \ldots LV_{\text{exit}}(\ell) \ldots \]

\[ LV_{\text{exit}}(\ell) = \ldots LV_{\text{entry}}(\ell) \ldots \]

All-paths/some-path: find greatest or least fixpoint solution to the equations

- \textit{lfp} (least fixpoint) for \( \exists \) path dependence

- \textit{gfp} (greatest fixpoint) for \( \forall \) path dependence

Combining flows from several blocks into one:

- Use \( \sqcup \) when computing least fixpoint (some-path properties)

- Use \( \sqcap \) when computing greatest fixpoint (all-path properties)
SEVERAL APPROACHES TO DATA FLOW ANALYSIS

- Data flow equations over a lattice (what we just saw)
- The “kill/gen” approach to data flow equations (a traditional compiler-writer’s approach)
- Constraint based analysis
- Monotone frameworks (unified lattice-theoretic viewpoint; notationally complex)
- Type and effect systems
- Abstract interpretation
“KILL/GEN” DATA FLOW EQUATIONS

For a future analysis $AN$:

$$AN_{\text{entry}}(\ell) = AN_{\text{exit}}(\ell) \setminus kill_{AN}(B^\ell) \sqcup gen_{AN}(B^\ell)$$

For a past analysis $AN$:

$$AN_{\text{exit}}(\ell) = AN_{\text{entry}}(\ell) \setminus kill_{AN}(B^\ell) \sqcup gen_{AN}(B^\ell)$$

Idea, reasoning:

- $kill_{AN}(B^\ell)$ expresses the data flow information that is over-written by statement $B^\ell$

- $gen_{AN}(B^\ell)$ expresses the new data flow information that is added by statement $B^\ell$

Example for live variable analysis: statement $[x:=y+z]^3$ will

- Generate $\{y,z\}$, so $gen_{LV}([x:=y+z]^3) = \{y, z\}$

- Kill $x$, so $kill_{LV}([x:=y+z]^3) = \{x\}$
MORE CONCRETELY: LIVE VARIABLE FLOW EQUATIONS

\[ y := x \] \[ z := 1 \] \text{ while } [y > 1] \text{ do } ([z := z \times y]; [y := y - 1]; [y := 0];

\[
\begin{align*}
LV_{entry}(1) &= LV_{exit}(1) \setminus \{y\} \cup \{x\} \\
LV_{entry}(2) &= LV_{exit}(2) \setminus \{z\} \\
LV_{entry}(3) &= LV_{exit}(3) \cup \{y\} \\
LV_{entry}(4) &= LV_{exit}(4) \setminus \{z\} \cup \{y, z\} \\
LV_{entry}(5) &= LV_{exit}(5) \setminus \{y\} \cup \{y\} \\
LV_{entry}(6) &= LV_{exit}(6) \setminus \{y\} \\
LV_{exit}(1) &= LV_{entry}(2) \\
LV_{exit}(2) &= LV_{entry}(3) \\
LV_{exit}(3) &= LV_{entry}(4) \cup LV_{entry}(6) \\
LV_{exit}(4) &= LV_{entry}(5) \\
LV_{exit}(5) &= LV_{entry}(3) \\
LV_{exit}(6) &= \emptyset \\
\end{align*}
\]

Examples: \( kill_{LV}([z := z \times y]) = \{z\} \) and \( gen_{LV}([z := z \times y]) = \{y, z\} \)
GENERAL DATA FLOW EQUATIONS: LIVE VARIABLES

\[
LV_{exit}(\ell) = \begin{cases} 
\emptyset & \text{if } [B]^{\ell} \text{ a final block} \\
\bigcup \{LV_{entry}(\ell') \mid \ell' \to \ell \text{ in flow chart} \} & \text{otherwise}
\end{cases}
\]

\[
LV_{entry}(\ell) = (LV_{exit}(\ell) \setminus kill_{LV}(B^{\ell})) \cup gen_{LV}(B^{\ell})
\]

where \( B^{\ell} \) is a block

▶ A future analysis, thus data flows backwards (from \( LV_{exit} \) to \( LV_{entry} \))
▶ An \( \exists \) path analysis, thus \( \text{lfp} \) and use \( \bigcup \) to merge branches

Some auxiliary definitions

\[
kill_{LV}([x := a]^{\ell}) = \{x\}
\]
\[
kill_{LV}([\text{skip}]^{\ell}) = \emptyset
\]
\[
kill_{LV}([b]^{\ell}) = \emptyset
\]

\[
gen_{LV}([x := a]^{\ell}) = \text{FreeVariables}(a)
\]
\[
gen_{LV}([\text{skip}]^{\ell}) = \emptyset
\]
\[
gen_{LV}([b]^{\ell}) = \text{FreeVariables}(b)
\]
GENERAL FLOW EQUATIONS: REACHING DEFINITIONS

\[ RD_{\text{entry}}(\ell) = \begin{cases} 
\{(x, ?) \mid x \in \text{FreeVariables}(S)\} & \text{if } [B]^{\ell} \text{ initial block} \\
\cup \{RD_{\text{exit}}(\ell') \mid \ell' \rightarrow \ell \text{ in flow chart}\} & \text{otherwise}
\end{cases} \]

\[ RD_{\text{exit}}(\ell) = (RD_{\text{entry}}(\ell) \setminus \text{kill}_{RD}(B^\ell)) \cup \text{gen}_{RD}(B^\ell) \]

where \( B^\ell \) is a block

▶ A past analysis, thus data flows forwards (from \( RD_{\text{entry}} \) to \( RD_{\text{exit}} \))

▶ An \( \exists \) path analysis, thus \( \text{lfp} \) and use \( \cup \) to merge branches

Some auxiliary definitions

\[ \text{kill}_{RD}([x := a]^\ell) = \{(x, ?)\} \cup \{(x, \ell') \mid \exists \text{ assignment } [x := \ldots]^\ell'\} \]

\[ \text{kill}_{RD}([\text{skip}]^\ell) = \emptyset \]

\[ \text{kill}_{RD}([b]^\ell) = \emptyset \]

\[ \text{gen}_{RD}([x := a]^\ell) = \{(x, \ell)\} \]

\[ \text{gen}_{RD}([\text{skip}]^\ell) = \emptyset \]

\[ \text{gen}_{RD}([b]^\ell) = \emptyset \]
CONSTRUCT SYSTEMS

Express flow equations in terms of set containments. *LV* example:

\[
[y:=x]^{1};[z:=1]^{2}; \text{while } [y>1]^{3} \text{ do } ([z:=z*y]^{4};[y:=y-1]^{5});[y:=0]^{6};
\]

\[
\text{LV}_{\text{entry}}(1) \supseteq \text{LV}_{\text{exit}}(1) \setminus \{y\} \quad \text{LV}_{\text{exit}}(1) \supseteq \text{LV}_{\text{entry}}(2) \\
\text{LV}_{\text{entry}}(1) \supseteq \{x\} \quad \text{LV}_{\text{exit}}(2) \supseteq \text{LV}_{\text{entry}}(3) \\
\text{LV}_{\text{entry}}(2) \supseteq \text{LV}_{\text{exit}}(2) \setminus \{z\} \\
\text{LV}_{\text{exit}}(2) \supseteq \text{LV}_{\text{entry}}(3) \\
\text{LV}_{\text{entry}}(3) \supseteq \text{LV}_{\text{exit}}(3) \\
\text{LV}_{\text{entry}}(3) \supseteq \{y\} \\
\text{LV}_{\text{exit}}(3) \supseteq \text{LV}_{\text{entry}}(4) \\
\text{LV}_{\text{entry}}(4) \supseteq \text{LV}_{\text{exit}}(4) \\
\text{LV}_{\text{exit}}(4) \supseteq \text{LV}_{\text{entry}}(5) \\
\text{LV}_{\text{entry}}(4) \supseteq \{y, z\} \\
\text{LV}_{\text{exit}}(5) \supseteq \text{LV}_{\text{entry}}(6) \\
\text{LV}_{\text{entry}}(5) \supseteq \{y\} \\
\text{LV}_{\text{exit}}(5) \supseteq \text{LV}_{\text{entry}}(3) \\
\text{LV}_{\text{entry}}(6) \supseteq \text{LV}_{\text{exit}}(6) \setminus \{y\}
\]

Exactly equivalent in this context. More generally: constraints can express more sophisticated flow analyses that are hard to describe by equations.
SEMANTIC CORRECTNESS, OR “SAFETY”

To show: that what the analysis says, is actually true of any computation.

- Starting point: the semantics of the programming language.

- Given a program $S$ and an initial store $\sigma$, the semantics defines the set of possible (finite or infinite) computations

  $\langle S, \sigma \rangle \rightarrow \langle S_1, \sigma_1 \rangle \rightarrow \langle S_2, \sigma_2 \rangle \rightarrow \ldots$

- Given: an analysis $AN$ of one (arbitrary) program

- Needed: a (logical and natural) connection between
  
  - the result of the analysis; and
  
  - the program’s possible computations

This is the start of the field:

Semantics-based program manipulation